Assessed coursework: Integrating R and C++

Cecina Babich Morrow

2024-01-29

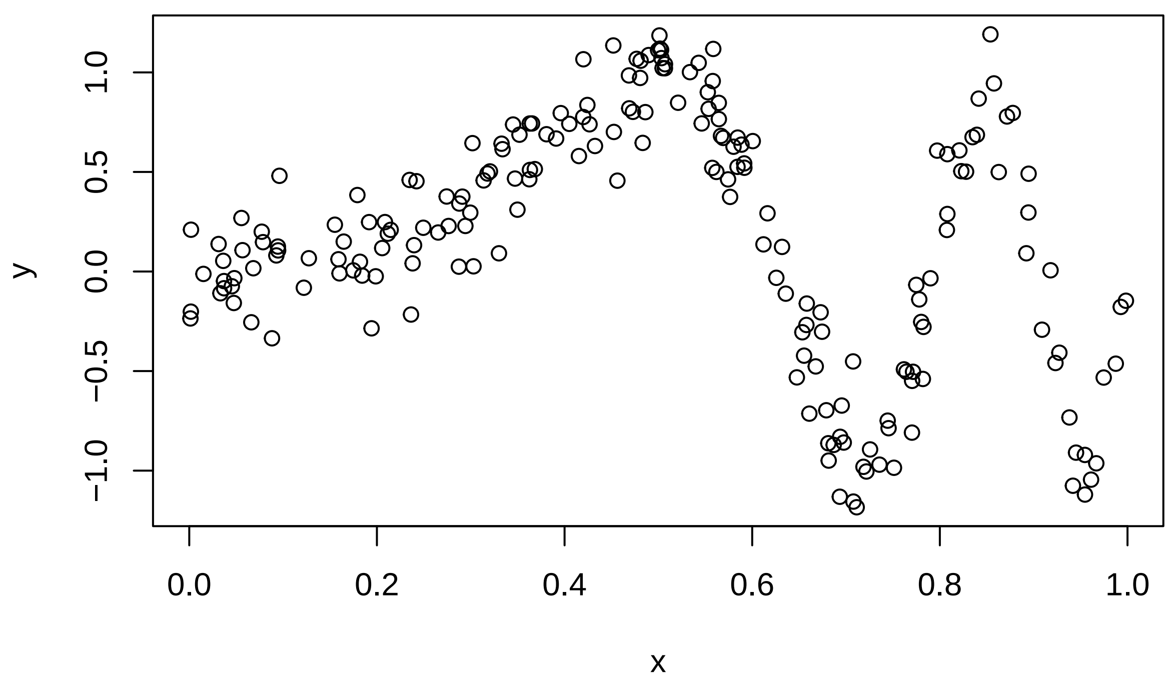
Adaptive kernel regression smoothing

We will consider data generated from the model

$$y_i = \sin(\alpha \pi x^3) + z_i$$

where $z_i \sim N(0, \sigma^2)$, $i \in \{1, ..., n\}$.

```
set.seed(998)
# n = 200
nobs <- 200
x <- runif(nobs)
# alpha = 4, sigma = 0.2
y <- sin(4*pi*x^3) + rnorm(nobs, 0, 0.2)
plot(x, y)
```



We want to model this data using a kernel regression smoother (KRS) by estimating $\mu(x) = \mathbb{E}(y|x)$. The KRS estimator is given by

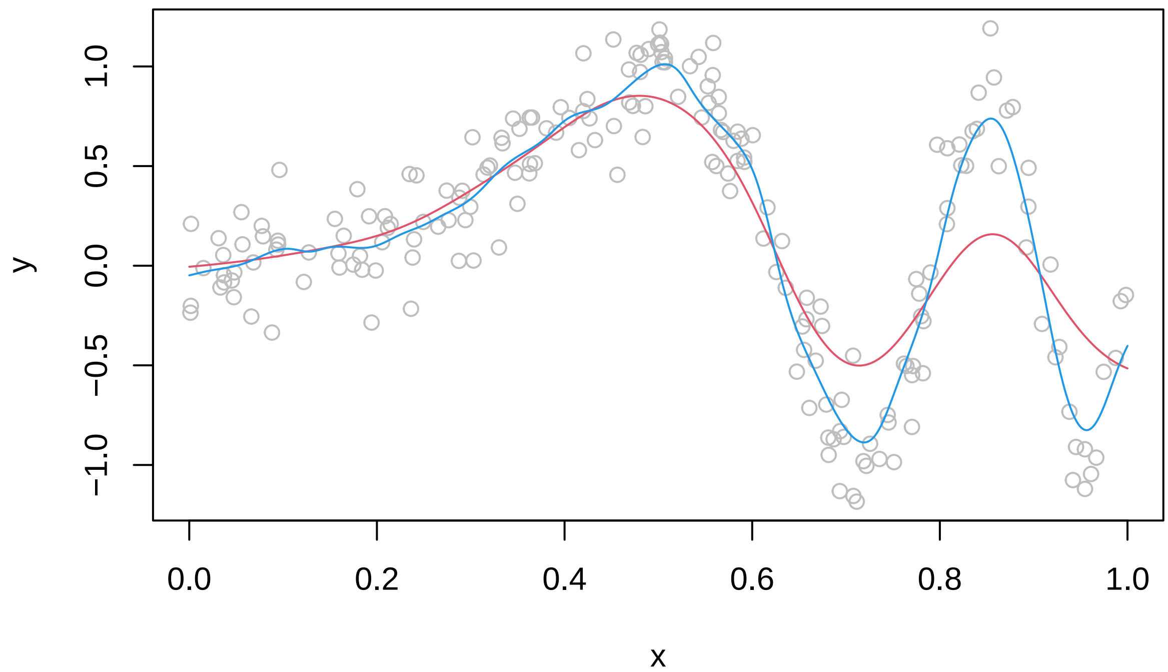
$$\hat{\mu}(x) = \frac{\sum_{i=1}^{n} \kappa_\lambda(x, x_i) y_i}{\sum_{i=1}^{n} \kappa_\lambda(x, x_i)}$$

where κ is a kernel function with bandwidth $\lambda > 0$. The following R function uses the Gaussian kernel function with variance λ^2:

```
meanKRS <- function(y, x, x0, lam){

 n <- length(x)
 n0 <- length(x0)

 out <- numeric(n0)
 for(ii in 1:n0){
  out[ii] <- sum( dnorm(x, x0[ii], lam) * y ) / sum( dnorm(x, x0[ii], lam) )
 }

 return( out )
}
```

We can compare the performance of the KRS estimator with different bandwidths:

```
xseq <- seq(0, 1, length.out = 1000)
muSmoothLarge <- meanKRS(y = y, x = x, x0 = xseq, lam = 0.06)
muSmoothSmall <- meanKRS(y = y, x = x, x0 = xseq, lam = 0.02)
plot(x, y, col = "grey")
lines(xseq, muSmoothLarge, col = 2)
lines(xseq, muSmoothSmall, col = 4)
```



Q1a

We want to write a C++ version of the `meanKRS` function. The function is as follows:

```
#include <R.h>
#include <Rinternals.h>
#include <Rmath.h>

SEXP meanKRS_C(SEXP y_vec, SEXP x_vec, SEXP x0_vec, SEXP lambda_param)
{

  int n = length(x_vec);
  int n0 = length(x0_vec);
  SEXP out = PROTECT(allocVector(REALSXP, n0));

  double *y = REAL(coerceVector(y_vec, REALSXP));
  double *x = REAL(coerceVector(x_vec, REALSXP));
  double *x0 = REAL(coerceVector(x0_vec, REALSXP));
  double lambda = REAL(lambda_param)[0];

  for (int i = 0; i < n0; i++)
  {
    double sum_dens_norm_y = 0;
    double sum_dens_norm = 0;
```

```
    for (int j = 0; j < n; j++)
    {
      double dens_norm = dnorm(x[j], x0[i], lambda, 0);
      sum_dens_norm_y += dens_norm * y[j];
      sum_dens_norm += dens_norm;
    }

    REAL(out)[i] = sum_dens_norm_y / sum_dens_norm;
  }

  UNPROTECT(1);

  return out;
}
```

We can compile the code and then load the function as follows:

```
# Compile the code
system(paste0("R CMD SHLIB ", here("portfolios/02_interfacing_r_with_c++/meanKRS_C.c")))
# Load the binary code
dyn.load(here("portfolios/02_interfacing_r_with_c++/meanKRS_C.so"))
# Check if the function is loaded
is.loaded("meanKRS_C")
```

```
## [1] TRUE
```

Next, we can call it using `.Call` and compare the results with the R function:

```
c_smooth_test <- .Call("meanKRS_C", y = y, x = x, x0 = xseq, lambda = 0.06)

# Compare with results of R function
all.equal(muSmoothLarge, c_smooth_test)
```

```
## [1] TRUE
```

```
# Compare computing time
krs_R <- function() meanKRS(y = y, x = x, x0 = xseq, lam = 0.06)
krs_C <- function() .Call("meanKRS_C", y = y, x = x, x0 = xseq, lambda = 0.06)
library(microbenchmark)
microbenchmark(krs_R(), krs_C(), times = 500)
```

```
## Unit: milliseconds
##     expr       min       lq     mean    median        uq       max neval
##  krs_R() 10.878803 10.96635 11.22733 11.005920 11.050527 15.071634   500
##  krs_C()  3.087495  3.11786  3.12825  3.128111  3.134812  3.745362   500
```

The C version of the function is over 3.5 times faster than the R version on average.

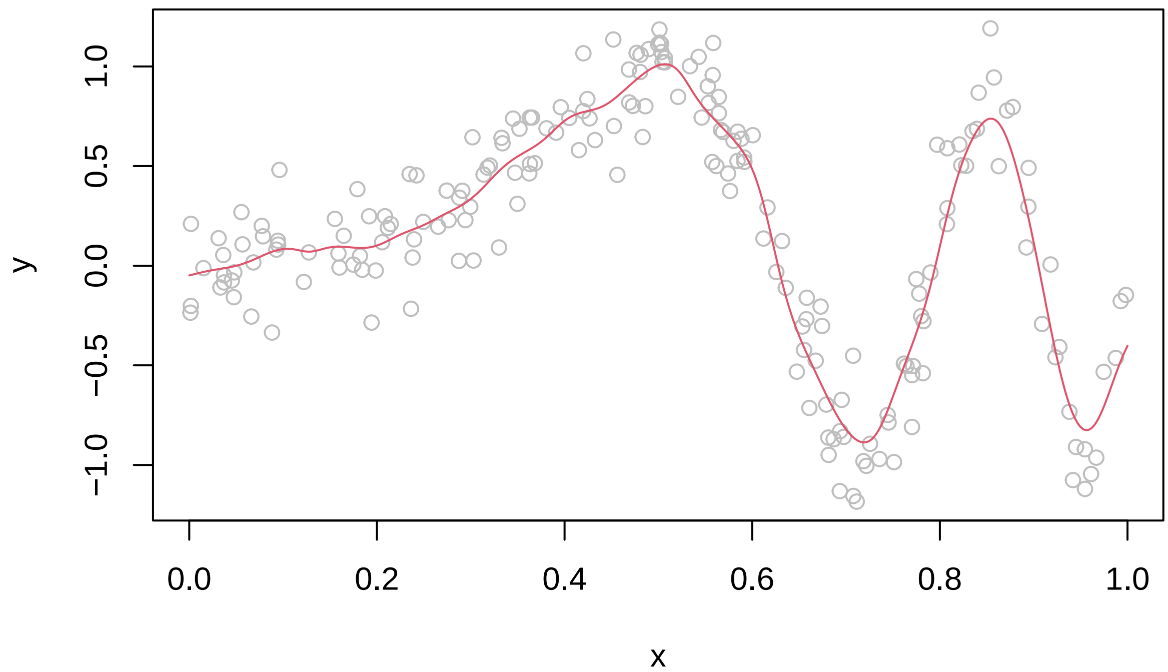
Q1b

We now want to implement k-fold cross-validation for selecting the bandwidth λ. We can first do so in R:

```r
krsCV <- function(y, x, k, lam_seq){

  n <- length(x)
  groups <- sample(rep(1:k, length.out = n), size = n)

  mse_table <- data.frame(lambda = rep(lam_seq, each = k),
                          fold = rep(1:k, length(lam_seq)),
                          mse = NA)
  for (lambda in lam_seq) {
    for (i in 1:k) {
      # Set up training and testing sets
      x_train <- x[groups != i]
      y_train <- y[groups != i]
      x_test <- x[groups == i]
      y_test <- y[groups == i]

      # Fit the model on the training set and get values for x_test
      mu_pred <- meanKRS(y = y_train, x = x_train, x0 = x_test, lam = lambda)

      # Calculate MSE on the testing set
      mse <- mean((mu_pred - y_test)^2)

      mse_table$mse[mse_table$lambda == lambda & mse_table$fold == i] <- mse
    }

  }

  mean_mse <- mse_table %>%
    group_by(lambda) %>%
    summarise(mean_mse = mean(mse))

  best_lambda <- mean_mse$lambda[which.min(mean_mse$mean_mse)]

  return(best_lambda)
}
```

Using this R function, we can find the value of λ with the lowest average mean squared error (MSE) over 5-fold cross-validation:

```r
best_lambda <- krsCV(y = y, x = x, k = 5, lam_seq = seq(0.01, 0.1, by = 0.01))
xseq <- seq(0, 1, length.out = 1000)
mu_best_lambda <- meanKRS(y = y, x = x, x0 = xseq, lam = best_lambda)
plot(x, y, col = "grey")
lines(xseq, mu_best_lambda, col = 2)
```



We can now write an equivalent function in C++ to compare results and performance:

Q2

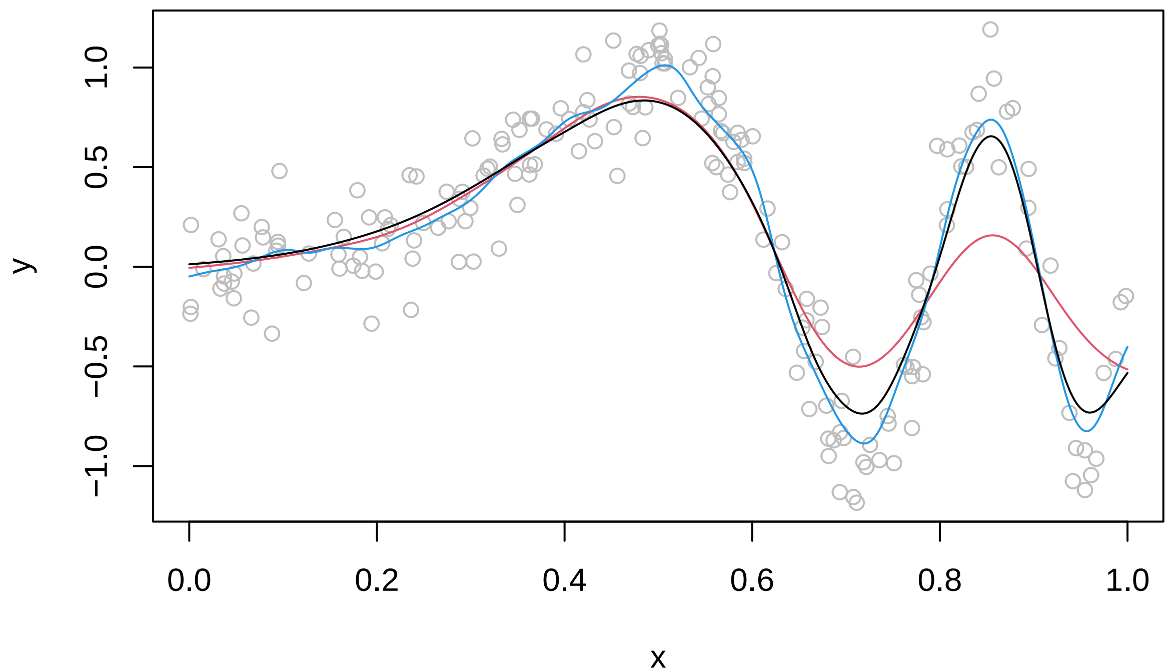
The following R function allows the bandwidth to depend on x, i.e. $\lambda = \lambda(x)$:

```
mean_var_KRS <- function(y, x, x0, lam){

 n <- length(x)
 n0 <- length(x0)
 mu <- res <- numeric(n)

 out <- madHat <- numeric(n0)

 for(ii in 1:n){
  mu[ii] <- sum( dnorm(x, x[ii], lam) * y ) / sum( dnorm(x, x[ii], lam) )
 }

 resAbs <- abs(y - mu)
 for(ii in 1:n0){
  madHat[ii] <- sum( dnorm(x, x0[ii], lam) * resAbs ) / sum( dnorm(x, x0[ii], lam) )
 }

 w <- 1 / madHat
```

```
 w <- w / mean(w)

 for(ii in 1:n0){
  out[ii] <- sum( dnorm(x, x0[ii], lam * w[ii]) * y ) /
             sum( dnorm(x, x0[ii], lam * w[ii]) )
 }

 return( out )
}
```

We can see the results here:

```
xseq <- seq(0, 1, length.out = 1000)
muSmoothAdapt <- mean_var_KRS(y = y, x = x, x0 = xseq, lam = 0.06)
plot(x, y, col = "grey")
lines(xseq, muSmoothLarge, col = 2) # red
lines(xseq, muSmoothSmall, col = 4) # blue
lines(xseq, muSmoothAdapt, col = 1) # black
```



We now want to write a version of `mean_var_KRS` in C++ and compare the results and performance with the R function.